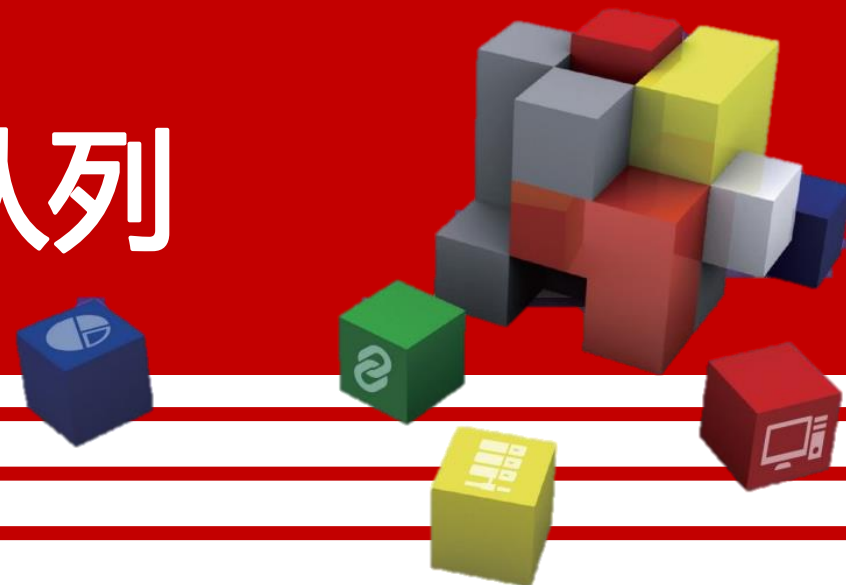


数据结构教程

(Python语言描述)

© 李春葆 主编 蒋林 李筱驰 副主编

第3章栈和队列





3.2 队列

提纲

CONTENTS

队列的定义

顺序队

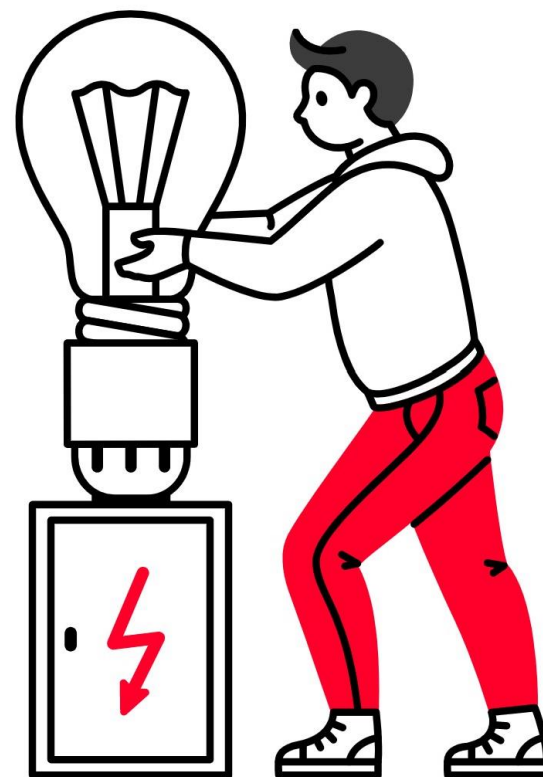
链 队

Python中的双端队列deque

队列的综合应用

双 端 队 列

优 先 队 列

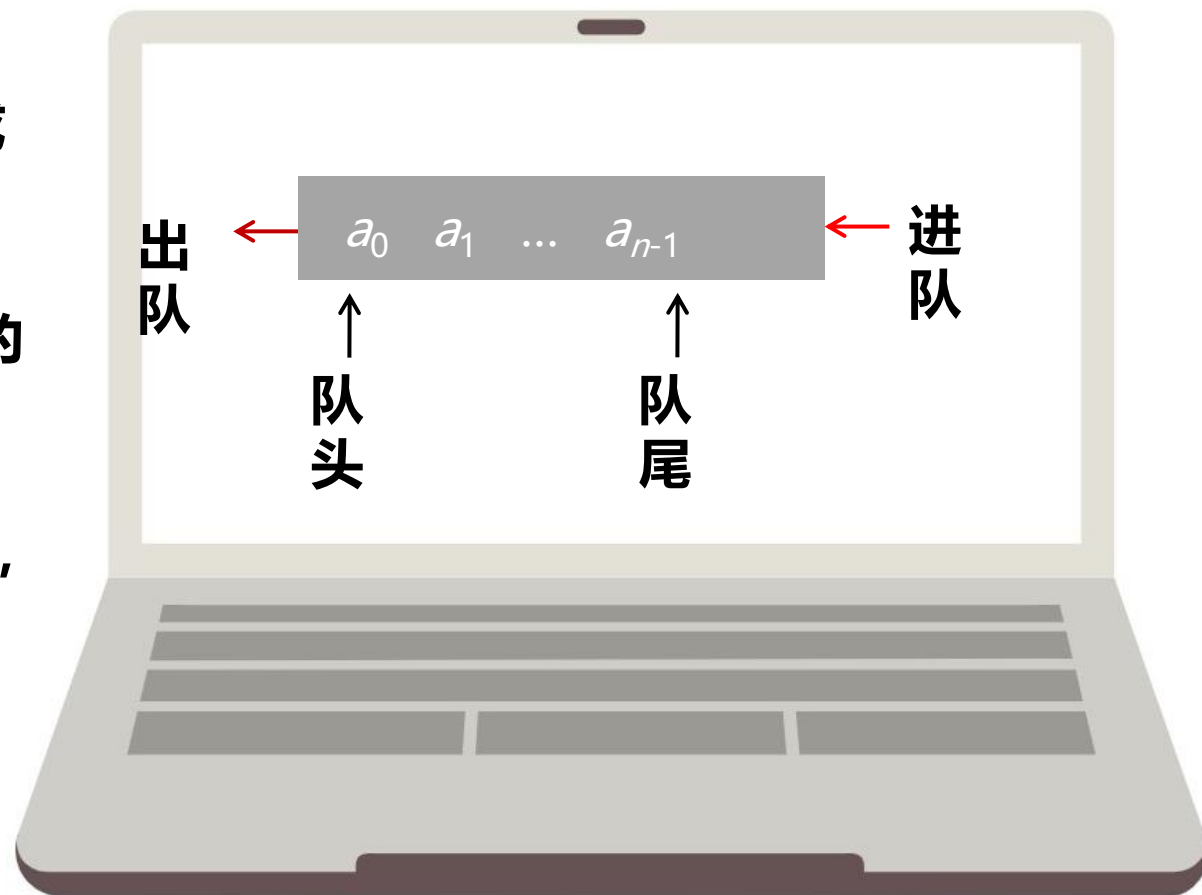




3.2.1 队列的定义

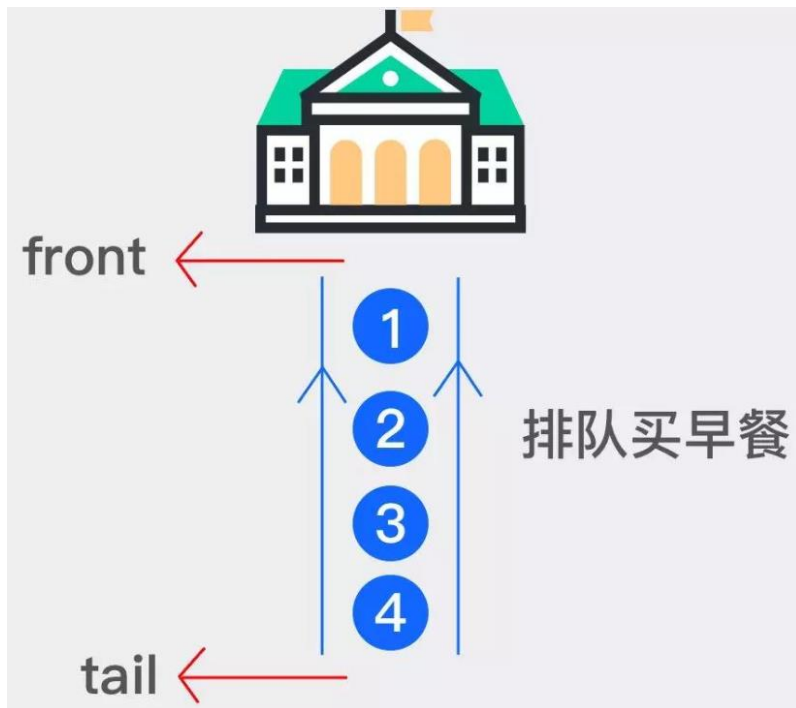
队列的定义

- **队列** (queue) 是一种只能在不同端进行插入或删除操作的线性表。
- 进行插入的一端称做**队尾 (rear)**，进行删除的一端称做**队头或队首 (front)**。
- 队列的插入操作通常称为**进队或入队 (push)**，队列的删除操作通常称为**出队或离队 (pop)**。





3.2.1 队列的定义



先买餐的人先出队





3.2.1 队列的定义



队列的**主要特点**:

- **先进先出**, 即先进队的元素先出队。
- 每次**进队**的元素作为**新队尾**元素,
每次**出队**的元素只能是**队头**的元素。
- 队列也称为**先进先出表**。



3.2.1 队列的定义

ADT Queue

{

数据对象:

$D = \{a_i \mid 0 \leq i \leq n-1, n \geq 0\}$

数据关系:

$R = \{r\}$

$r = \{ \langle a_i, a_{i+1} \rangle \mid a_i, a_{i+1} \in D, i = 0, \dots, n-2 \}$

基本运算:

empty(): 判断队列是否为空, 若队列为空, 返回真, 否则返回假。

push(e): 进队, 将元素e进队作为队尾元素。

pop(): 出队, 从队头出队一个元素。

gethead(): 取队头, 返回队头元素而不出队。

}

队列抽象数据类型 = 线性结构 + 队列的基本运算





3.2.1 队列的定义

【例3.10】若元素进队顺序为1234，能否得到3142的出队序列？

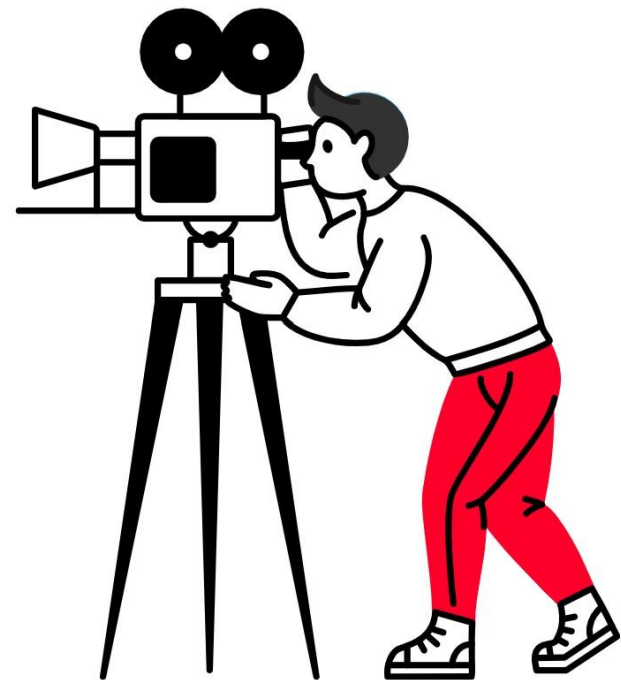
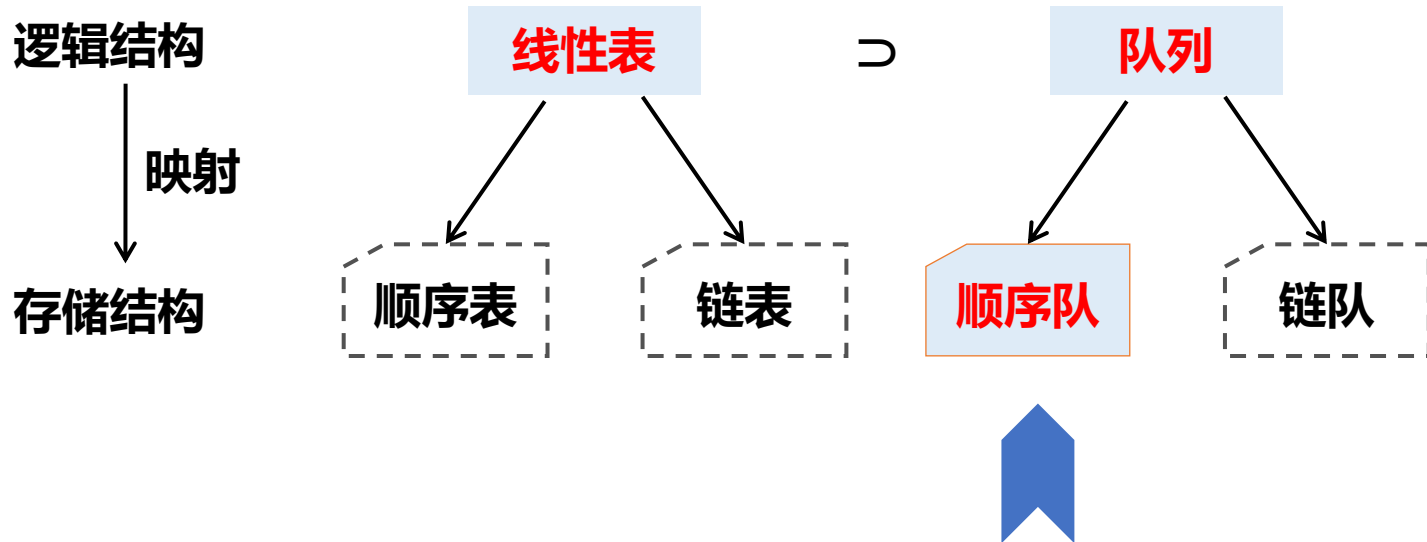


解：进队顺序为1234，则出队的顺序也为1234（先进先出），所以不能得到3142的出队序列。



3.2.2 队列的顺序存储结构及其基本运算算法实现

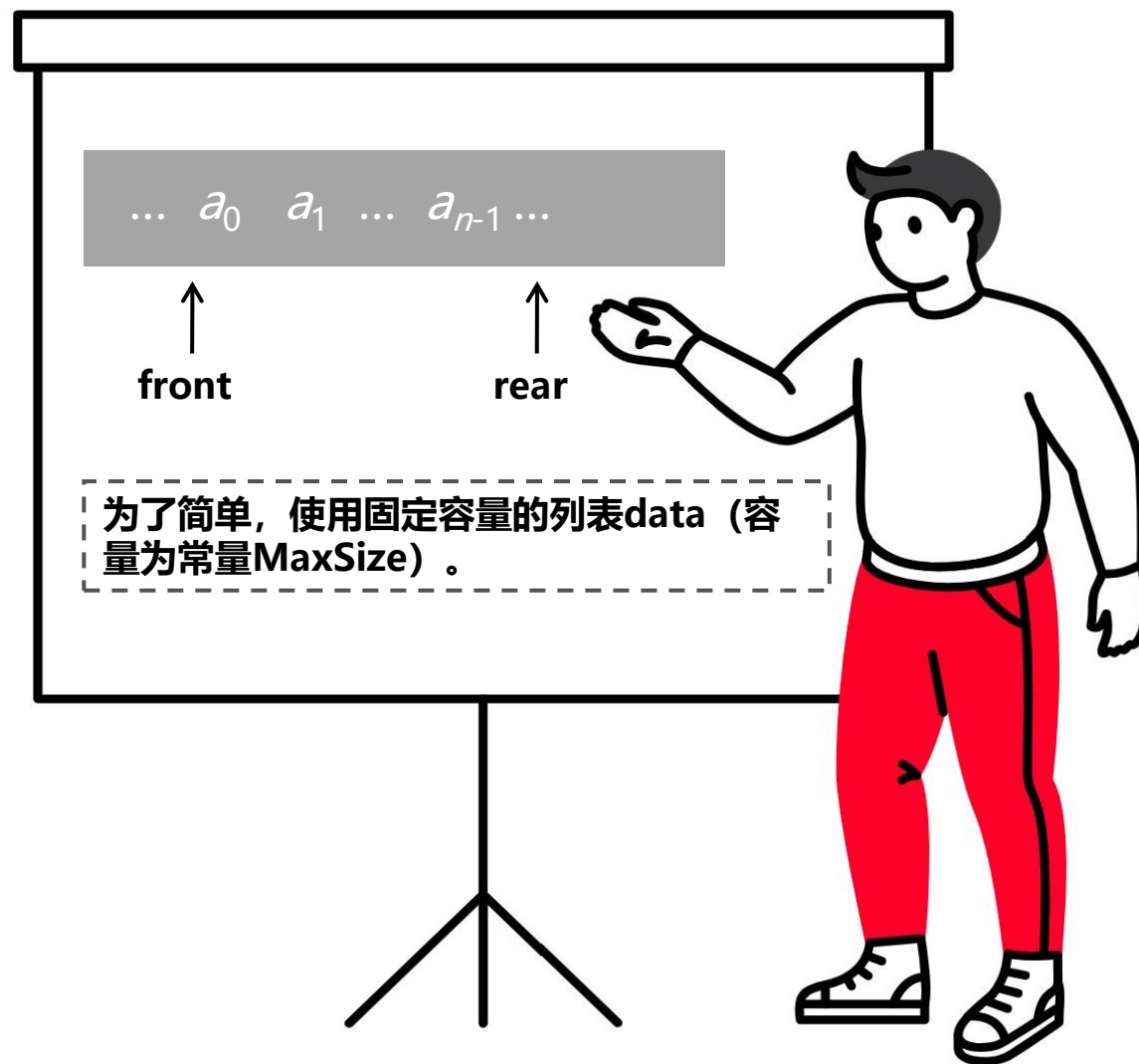
队列的实现方式





3.2.2 队列的顺序存储结构及其基本运算算法实现

- 用data列表来存放队列中元素。
- 队头指针为front（实际上是队头元素的前一个位置），
- 队尾指针为rear（正好是队尾元素的位置）。

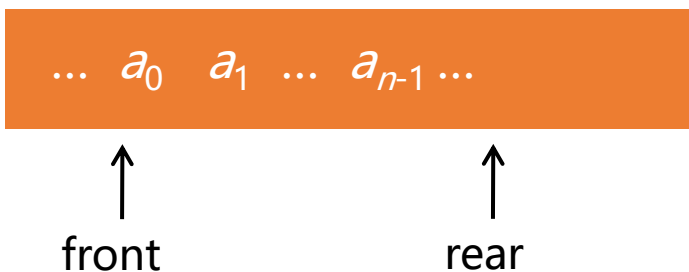




3.2.2 队列的顺序存储结构及其基本运算算法实现

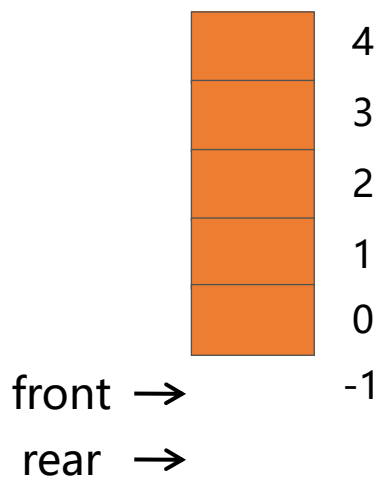
1. 非循环队列

- 初始时置front和rear均为-1 ($\text{front} = \text{rear}$)
- 元素进队, rear增加1
- 元素出队列, front增加1

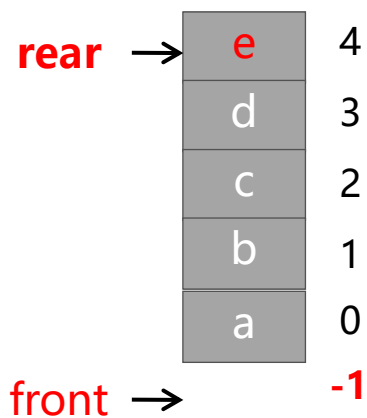




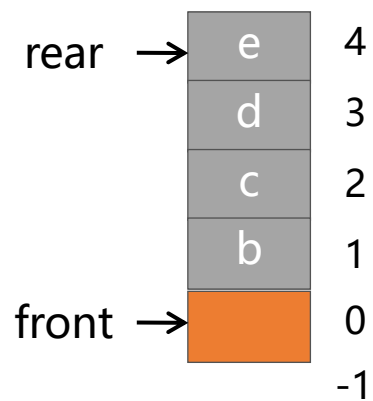
3.2.2 队列的顺序存储结构及其基本运算算法实现



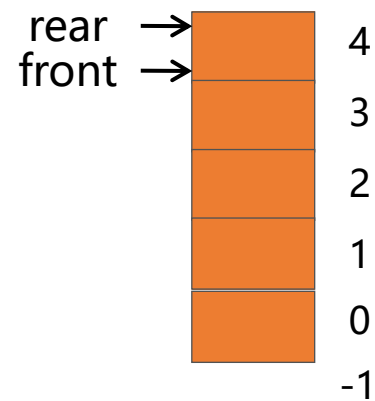
(a) 空队



(b) 5个元素进队



(c) 出队1次



(c) 出队4次



3.2.2 队列的顺序存储结构及其基本运算算法实现

初始时置front和rear均为-1 ($\text{front} = \text{rear}$)，该顺序队的四要素如下：

- **队空条件：** $\text{front} = \text{rear}$ 。
- **队满（上溢出）条件：** $\text{rear} = \text{MaxSize} - 1$ （因为每个元素进队都让rear增1，当rear到达最大下标时不能再增加。
- **元素e进队操作：**rear增1，将元素e放在该位置（进队的元素总是在尾部插入的）。
- **出队操作：**front增1，取出该位置的元素（出队的元素总是在队头出来的）。





3.2.2 队列的顺序存储结构及其基本运算算法实现



MaxSize=100

#假设容量为100

class SqQueue:

#非循环队列类

def __init__(self):

#构造方法

self.data=[None]*MaxSize

#存放队列中元素

self.front=-1

#队头指针

self.rear=-1

#队尾指针

#队列的基本运算算法

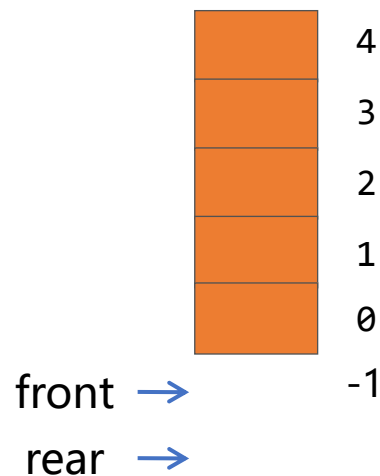


3.2.2 队列的顺序存储结构及其基本运算算法实现

非循环队列的基本运算算法

(1) 判断队列是否为空empty()

```
def empty(self):          #判断队列是否为空  
    return self.front == self.rear
```

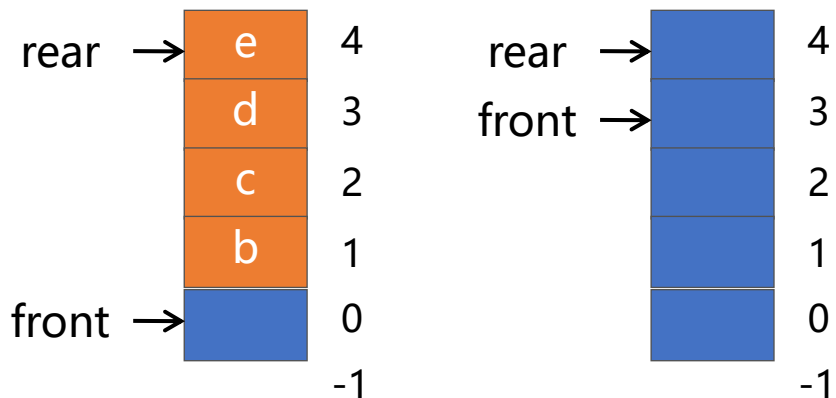




3.2.2 队列的顺序存储结构及其基本运算算法实现

(2) 进队push(e)

```
def push(self,e):  
    assert not self.rear==MaxSize-1    #元素e进队  
    self.rear+=1                        #检测队满  
    self.data[self.rear]=e
```

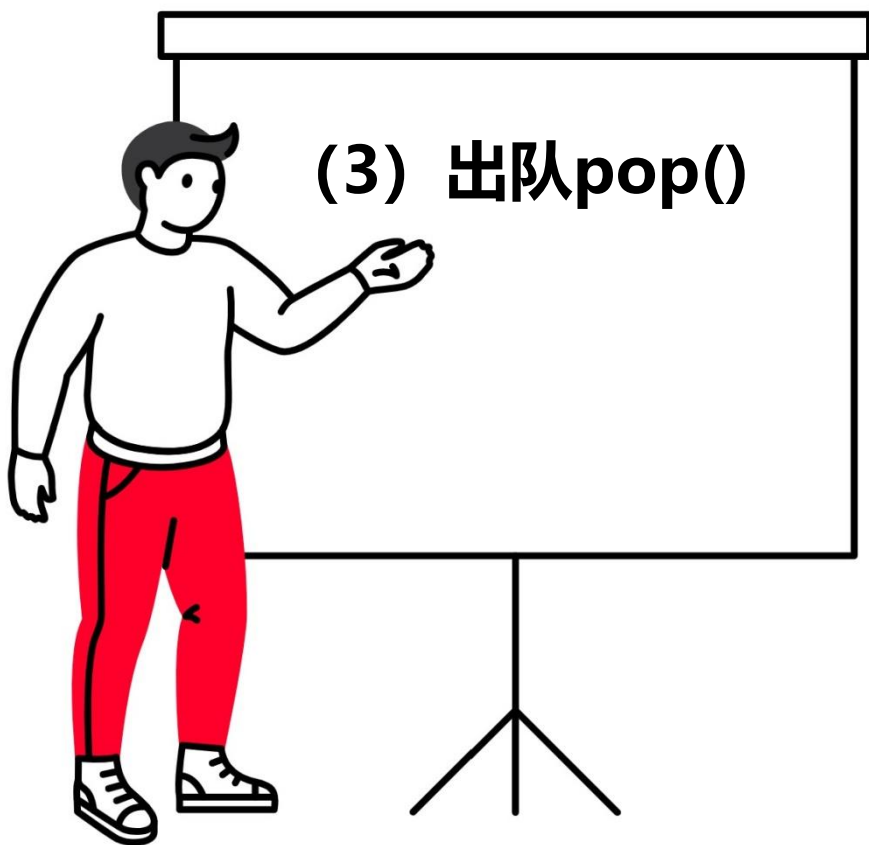


← 假溢出!





3.2.2 队列的顺序存储结构及其基本运算算法实现



```
def pop(self):  
    assert not self.empty()  
    self.front += 1  
    return self.data[self.front]
```

#出队元素
#检测队空



3.2.2 队列的顺序存储结构及其基本运算算法实现



(4) 取队头元素gethead()

```
def gethead(self):  
    assert not self.empty()  
    return self.data[self.front+1]
```

#取队头元素
#检测队空

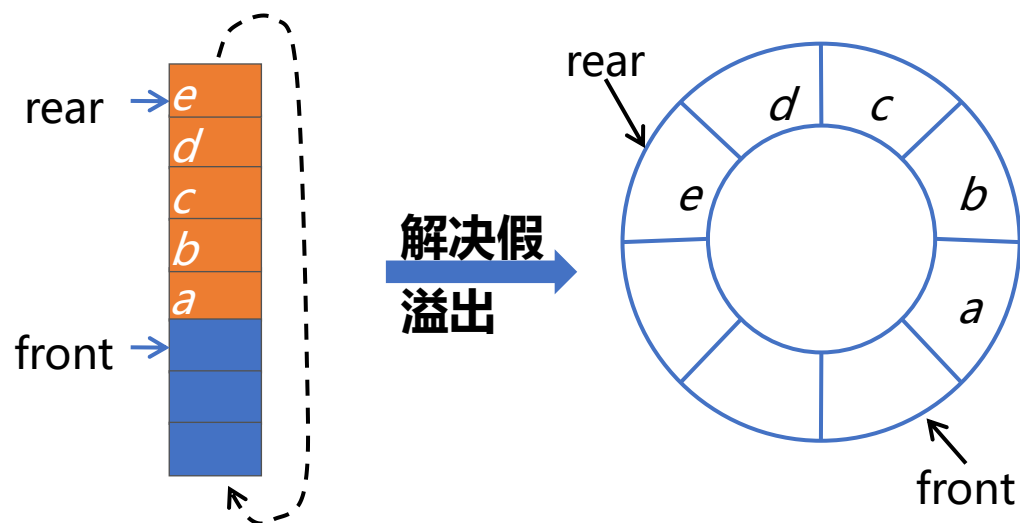


3.2.2 队列的顺序存储结构及其基本运算算法实现

2. 循环队列

- 把data数组的**前端**和**后端**连接起来，形成一个循环数组，即把存储队列元素的表从逻辑上看成一个环，称为**循环队列**（也称为环形队列）。

MaxSize=8

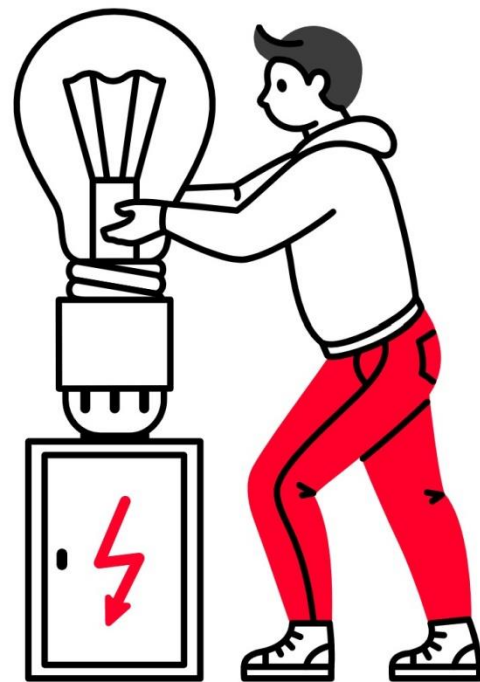




3.2.2 队列的顺序存储结构及其基本运算算法实现

- 循环队列**首尾**相连，当队尾指针 $\text{rear} = \text{MaxSize} - 1$ 时，再前进一个位置就应该到达0位置，这可以利用数学上的求余运算（%）实现：

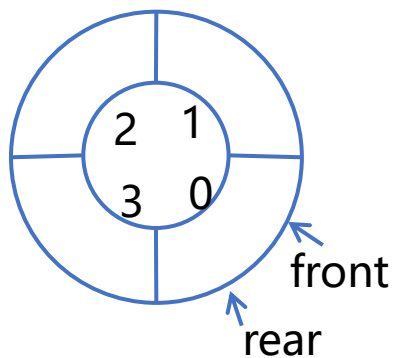
- 队首指针循环进1： $\text{front} = (\text{front} + 1) \% \text{MaxSize}$
- 队尾指针循环进1： $\text{rear} = (\text{rear} + 1) \% \text{MaxSize}$



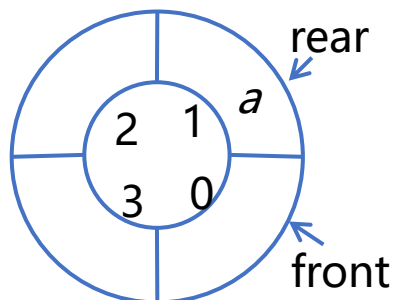


3.2.2 队列的顺序存储结构及其基本运算算法实现

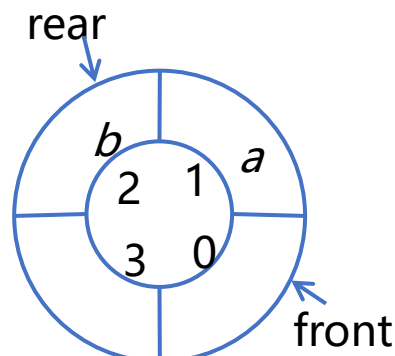
MaxSize=4, 初始front=rear=0



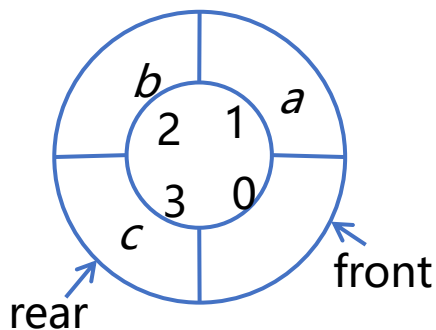
(a) 空队



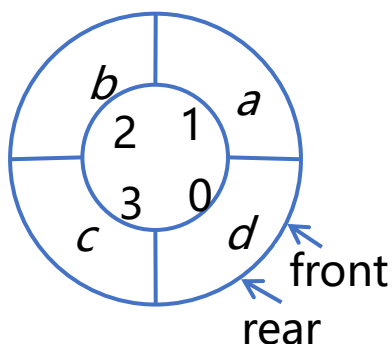
(b) a进队



(c) b进队



(d) c进队



(e) d进队





3.2.2 队列的顺序存储结构及其基本运算算法实现

- 顺序队（含循环队列和非循环队列）通过 $front$ 和 $rear$ 标识队列状态，一般是采用它们的相对值即 $|front - rear|$ 实现的。
- 若 $data$ 数组的容量为 m ，则队列的状态有 $m+1$ 种，分别是队空、队中有1个元素、队中有2个元素、...、队中有 m 个元素（队满）。
- $front$ 和 $rear$ 的取值范围均为 $0 \sim m-1$ ，这样 $|front - rear|$ 只有 m 个值。
- 显然 $m+1$ 种状态不能直接用 $|front - rear|$ 区分，因为必定有两种状态不能区分。
- 为此让队列中最多只有 $m-1$ 个元素，这样队列恰好只有 m 种状态了，就可以通过 $front$ 和 $rear$ 的相对值区分所有状态了。



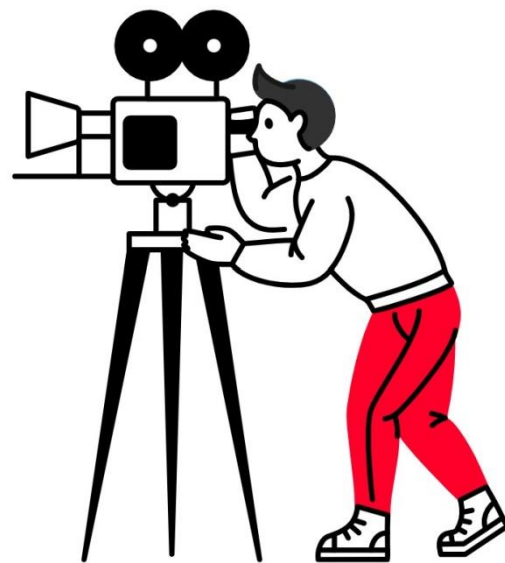


3.2.2 队列的顺序存储结构及其基本运算算法实现

在规定队中最多只有 $m-1$ 个元素时，设置队空条件仍然是 $\text{rear} == \text{front}$ 。当队列有 $m-1$ 个元素时一定满足 $(\text{rear} + 1) \% \text{MaxSize} == \text{front}$ 。

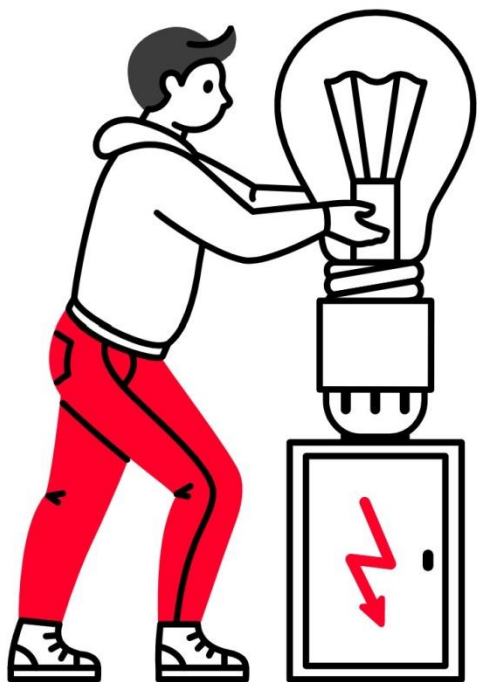
这样，循环队列在初始时置 $\text{front} = \text{rear} = 0$ ，其四要素如下：

- **队空条件：** $\text{rear} == \text{front}$ 。
- **队满条件：** $(\text{rear} + 1) \% \text{MaxSize} == \text{front}$ （相当于试探进队一次，若 rear 达到 front ，则认为队满了）。
- **元素e进队：** $\text{rear} = (\text{rear} + 1) \% \text{MaxSize}$ ，将元素 e 放置在该位置。
- **元素出队：** $\text{front} = (\text{front} + 1) \% \text{MaxSize}$ ，取出该位置的元素。





3.2.2 队列的顺序存储结构及其基本运算算法实现



循环队列类SqQueue

MaxSize=100

#全局变量，假设容量为100

class CSqQueue:

#循环队列类

def __init__(self):

#构造方法

self.data=[None]*MaxSize

#存放队列中元素

self.front=0

#队头指针

self.rear=0

#队尾指针

#队列的基本运算算法



3.2.2 队列的顺序存储结构及其基本运算算法实现

循环队列的基本运算算法

(1) 判断队列是否为空empty()

```
def empty(self):
```

#判断队列是否为空

```
    return self.front == self.rear
```





3.2.2 队列的顺序存储结构及其基本运算算法实现

(2) 进队push(e)

```
def push(self,e):
```

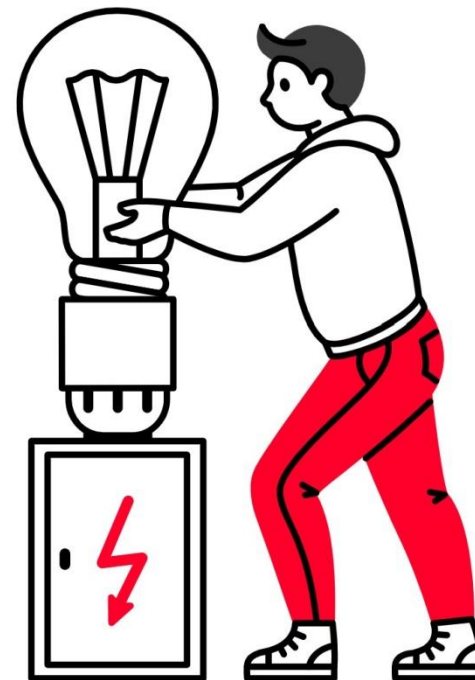
```
    assert (self.rear+1)%MaxSize!=self.front
```

```
    self.rear=(self.rear+1)%MaxSize
```

```
    self.data[self.rear]=e
```

#元素e进队

#检测队满





3.2.2 队列的顺序存储结构及其基本运算算法实现



(3) 出队pop()

```
def pop(self):                                #出队元素
    assert not self.empty()                   #检测队空
    self.front=(self.front+1)%MaxSize
    return self.data[self.front]
```



3.2.2 队列的顺序存储结构及其基本运算算法实现

(4) 取队头元素gethead()

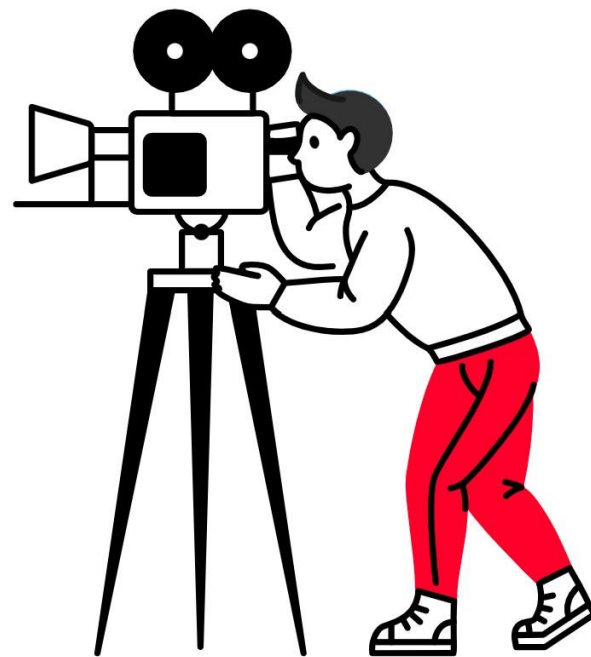
```
def gethead(self):                #取队头元素
    assert not self.empty()        #检测队空
    head=(self.front+1)%MaxSize    #求队头元素的位置
    return self.data[head]
```



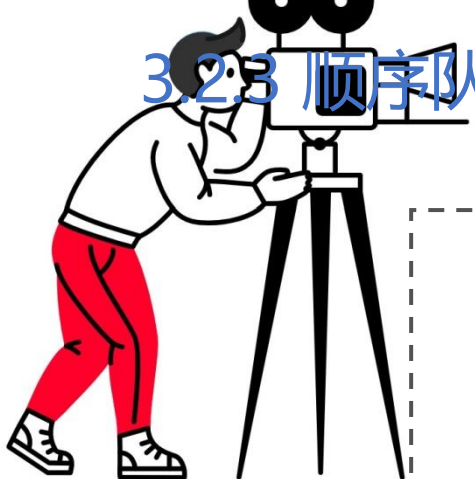


3.2.3 顺序队的应用算法设计示例

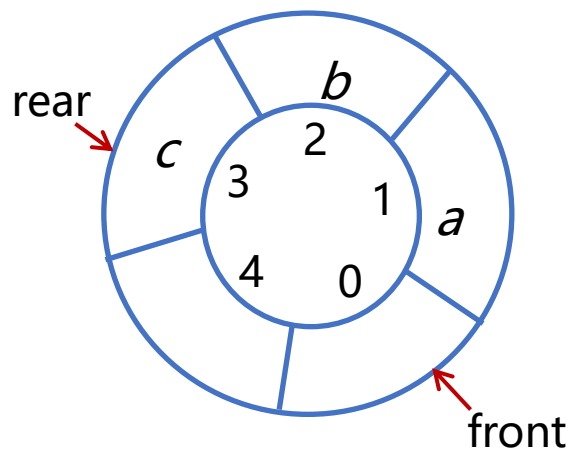
【例3.11】在CSqQueue循环队列类中增加一个求元素个数的算法size()。对于一个整数循环队列qu，利用队列基本运算和size()算法设计进队和出队第k ($k \geq 1$ ，队头元素的序号为1) 个元素的算法。



3.2.3 顺序队的应用算法设计示例



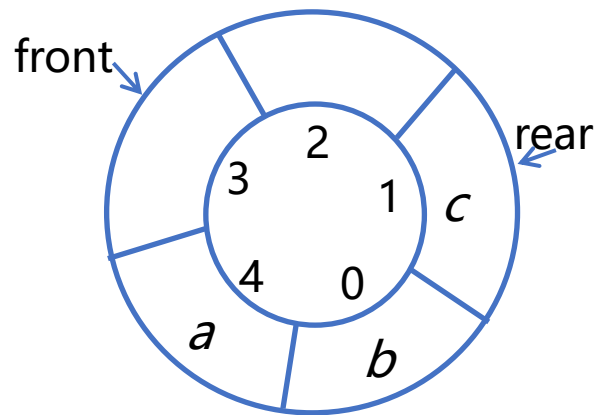
MaxSize=5



$$\text{cnt} = (\text{rear} - \text{front}) = 3$$

$$\text{cnt} = (\text{rear} - \text{front} + \text{MaxSize}) = 8 \times$$

$$\text{cnt} = (\text{rear} - \text{front} + \text{MaxSize}) \% \text{MaxSize} = 3 \checkmark$$



$$\text{cnt} = (\text{rear} - \text{front}) = -2 \times$$

$$\text{cnt} = (\text{rear} - \text{front} + \text{MaxSize}) = 3$$

$$\text{cnt} = (\text{rear} - \text{front} + \text{MaxSize}) \% \text{MaxSize} = 3 \checkmark$$



3.2.3 顺序队的应用算法设计示例

在CSqQueue循环队列类中增加size()算法如下:

```
def size(self):           #返回队中元素个数  
    return ((self.rear-self.front+MaxSize)%MaxSize)
```

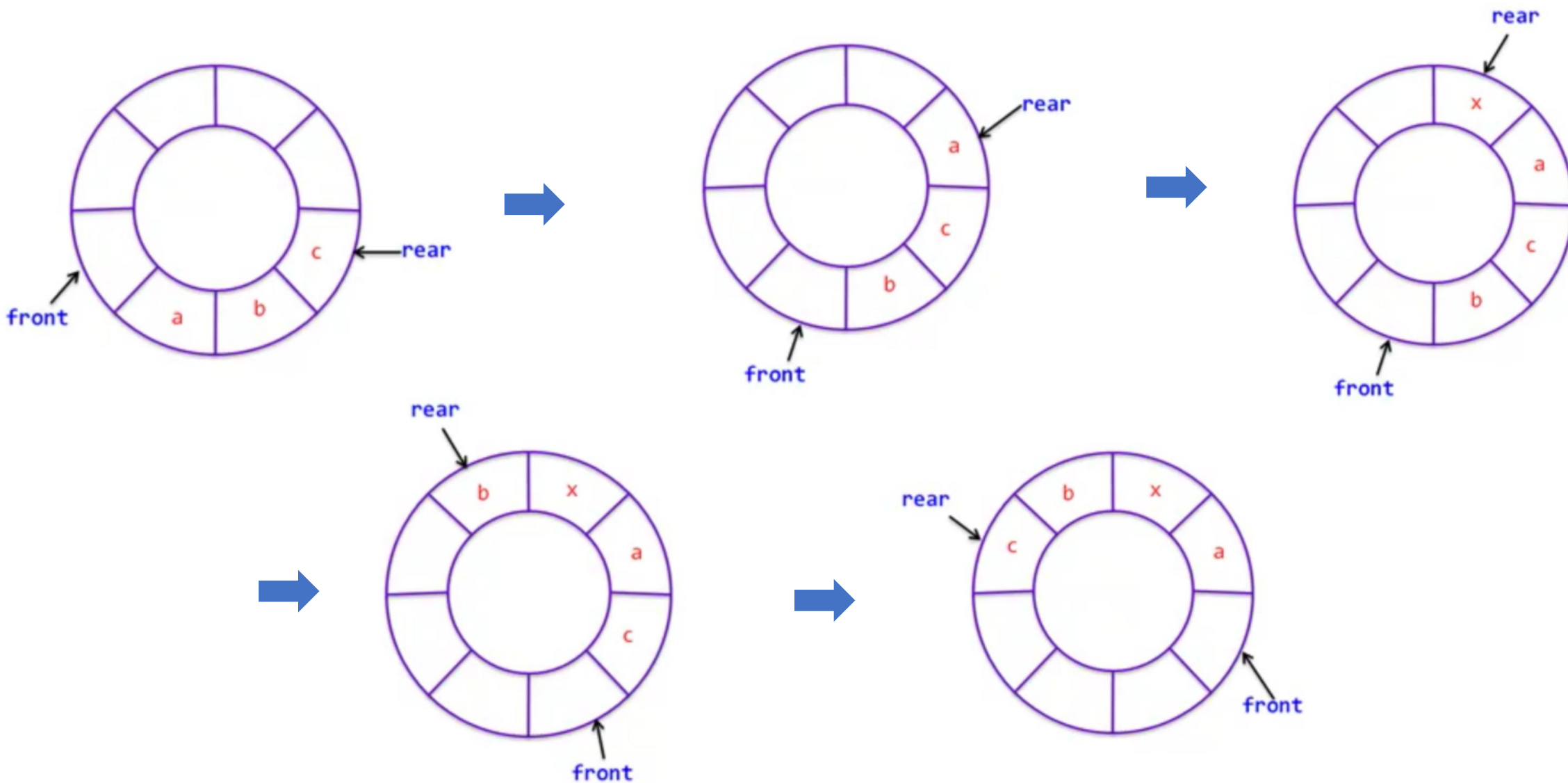




3.2.3 顺序队的应用算法设计示例

进队第 k ($k \geq 1$) 个元素 e

例如: $e = 'x'$, $k = 2$: $a\ b\ c \Rightarrow a\ x\ b\ c$





3.2.3 顺序队的应用算法设计示例

进队第 k ($k \geq 1$) 个元素 e 的算法:

```
def pushk(qu,k,e):  
    n=qu.size()  
    if k<1 or k>n+1:  
        return False  
    if k<=n:  
        for i in range(1,n+1):  
            if i==k:  
                qu.push(e)  
                x=qu.pop()  
                qu.push(x)  
            else: qu.push(e)  
        return True
```

#进队第 k 个元素 e

#参数 k 错误返回False

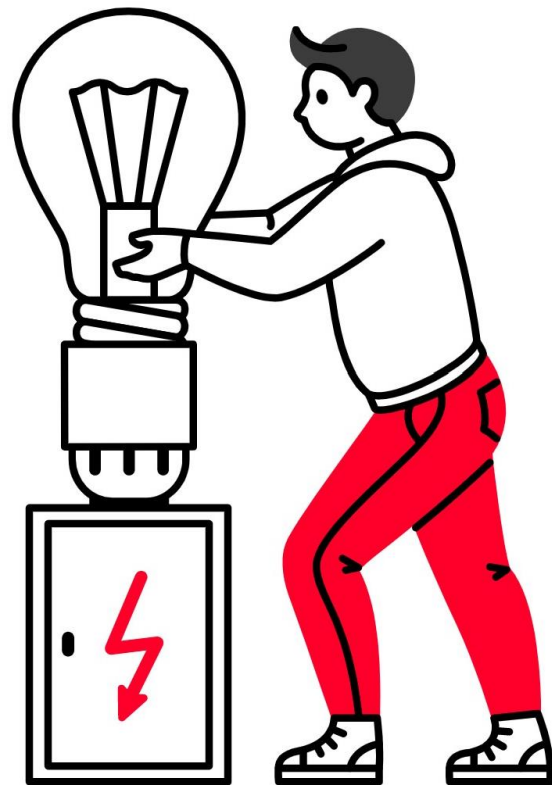
#循环处理队中所有元素

#将 e 元素进队到第 k 个位置

#出队元素 x

#进队元素 x

$k=n+1$ 时直接进队 e





3.2.3 顺序队的应用算法设计示例



出队第 k ($k \geq 1$) 个元素 e 的算法思路:

出队前 $k-1$ 个元素, 边出边进, 出队第 k 个元素 e , e 不进队, 将剩下的元素边出边进。

```
def popk(qu,k):  
    n=qu.size()  
    assert k>=1 and k<=n  
    for i in range(1,n+1):  
        x=qu.pop()  
        if i!=k: qu.push(x)  
        else: e=x  
    return e
```

#出队第 k 个元素

#检测参数 k 错误
#循环处理队中所有元素
#出队元素 x
#将非第 k 个元素进队
#取第 k 个出队的元素



3.2.3 顺序队的应用算法设计示例

【例3.12】对于循环队列来说，如果知道**队头指针**和**队列中元素个数**，则可以计算出**队尾指针**。也就是说，可以用队列中元素个数代替队尾指针。设计出这种循环队列的判队空、进队、出队和取队头元素的算法。

$$\text{count} = (\text{rear} - \text{front} + \text{MaxSize}) \% \text{MaxSize}$$



已知**front**、**count**，求**rear**：

$$\text{rear} = (\text{front} + \text{count}) \% \text{MaxSize}$$

已知**rear**、**count**，求**front**：

$$\text{front} = (\text{rear} - \text{count} + \text{MaxSize}) \% \text{MaxSize}$$





对应的循环队列类CSqQueue1



```
MaxSize=100                                     #全局变量，假设容量为100
class CSqQueue1:                                #本例循环队列类
    def __init__(self):                          #构造方法
        self.data=[None]*MaxSize                #存放队列中元素
        self.front=0                            #队头指针
        self.count=0                            #队中元素个数
```



3.2.3 顺序队的应用算法设计示例

#队列的基本运算算法

```
def empty(self):
```

```
    return self.count == 0
```

```
def push(self, e):
```

```
    rear1 = (self.front + self.count) % MaxSize;
```

```
    assert self.count != MaxSize
```

```
    rear1 = (rear1 + 1) % MaxSize
```

```
    self.data[rear1] = e
```

```
    self.count += 1
```

#判断队列是否为空

#元素e进队

#检测队满

#元素个数增1

方法中的局部变量





3.2.3 顺序队的应用算法设计示例

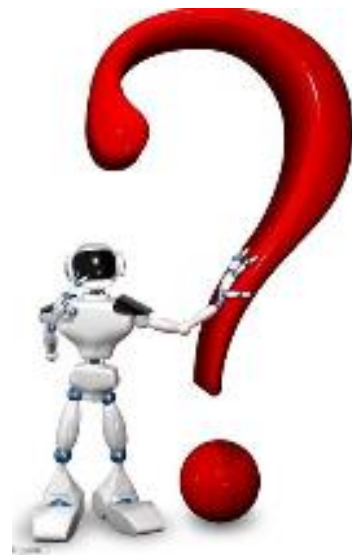
```
def pop(self):                                #出队元素
    assert not self.empty()                   #检测队空
    self.count-=1                             #元素个数减1
    self.front=(self.front+1)%MaxSize         #队头指针循环进1
    return self.data[self.front]
def gethead(self):                            #取队头元素
    assert not self.empty()                   #检测队空
    head=(self.front+1)%MaxSize               #求队头元素的位置
    return self.data[head]
```

说明

本例设计的循环队列中最多可保存MaxSize个元素。



3.2.3 顺序队的应用算法设计示例





3.2.4 队列的链式存储结构及其基本运算算法实现



队列的实现方式

逻辑结构

映射

存储结构

线性表

⊃

队列

顺序表

链表

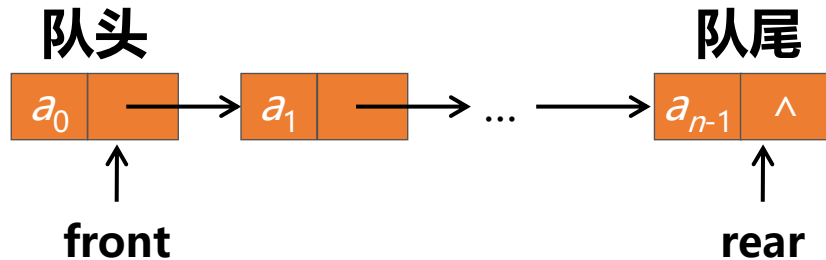
顺序队

链队





3.2.4 队列的链式存储结构及其基本运算算法实现

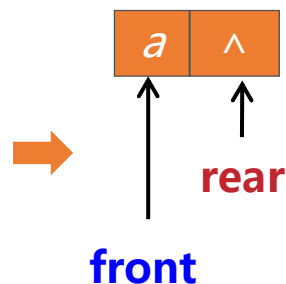


操作示例

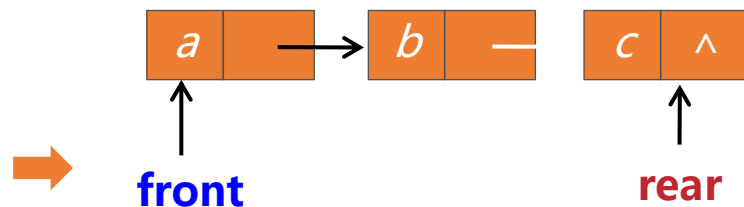
front=None

rear=None

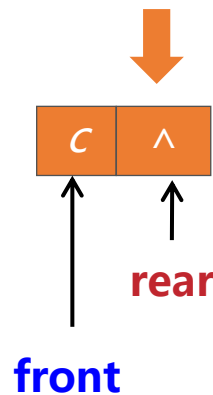
(a) 链队初态



(b) a进队



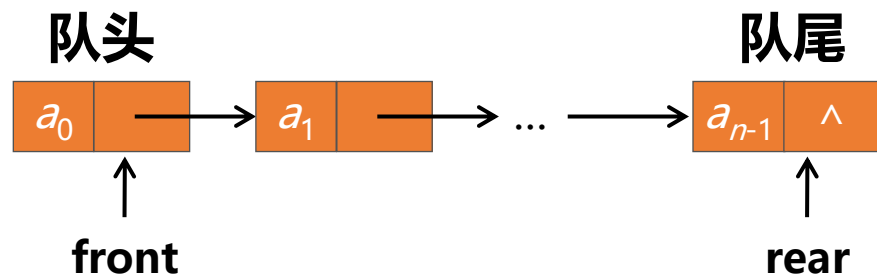
(c) b,c进队



(d) 出队两次



3.2.4 队列的链式存储结构及其基本运算算法实现



初始时置 $\text{front}=\text{rear}=\text{None}$ 。链队的四要素如下：

- **队空条件**: $\text{front}=\text{rear}==\text{None}$ ，不妨仅以 $\text{front}==\text{None}$ 作为队空条件。
- 由于只有内存溢出时才出现**队满**，通常**不考虑**这样的情况。
- **元素e进队操作**：在单链表尾部插入存放e的s结点，并让队尾指针指向它。
- **出队操作**：取出队首结点的data值并将其从链队中删除。





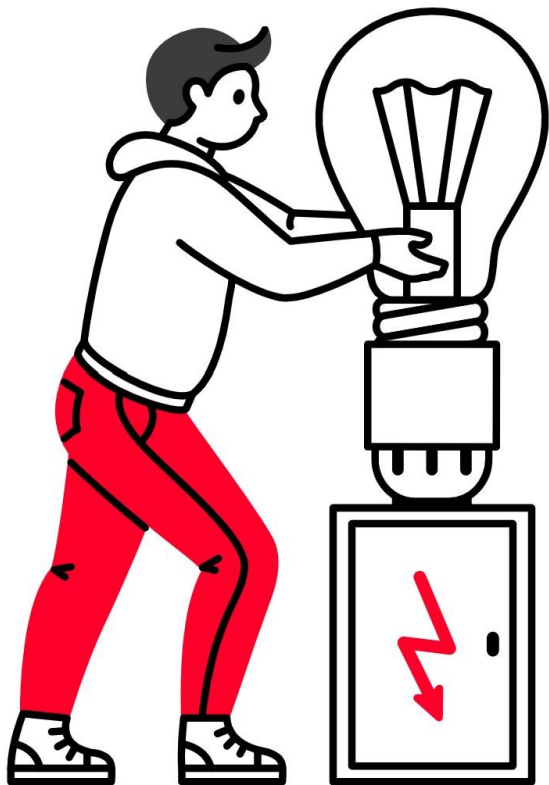
3.2.4 队列的链式存储结构及其基本运算算法实现



```
class LinkNode:           #链队结点类
    def __init__(self,data=None): #构造方法
        self.data=data       #data属性
        self.next=None       #next属性
```



3.2.4 队列的链式存储结构及其基本运算算法实现



链队类LinkQueue

```
class LinkQueue:  
    def __init__(self):  
        self.front=None  
        self.rear=None  
#队列的基本运算算法
```

```
#链队类  
#构造方法  
#队头指针  
#队尾指针
```



3.2.4 队列的链式存储结构及其基本运算算法实现



链队的基本运算算法

(1) 判断队列是否为空empty()

```
def empty(self):
```

#判断队是否为空

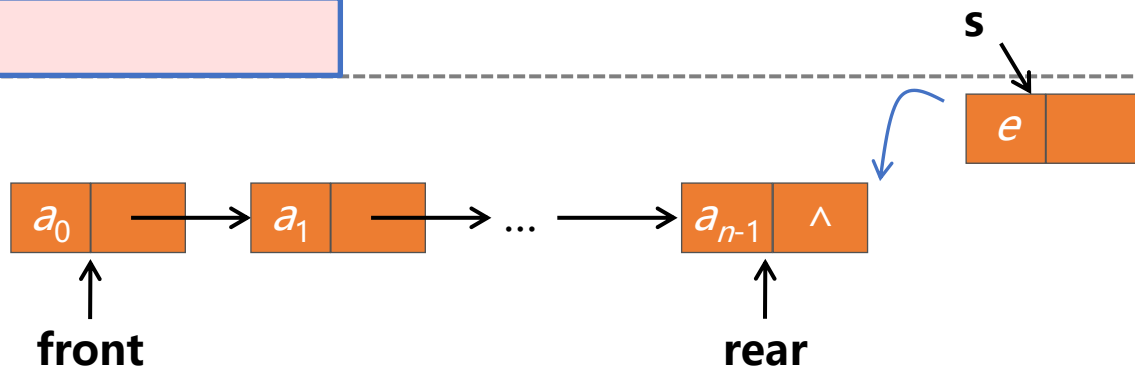
```
    return self.front == None
```



3.2.4 队列的链式存储结构及其基本运算算法实现

(2) 进队push(e)

```
def push(self,e):           #元素e进队
    s=LinkNode(e)          #新建结点s
    if self.empty():        #原链队为空
        self.front=self.rear=s
    else:                   #原链队不空
        self.rear.next=s    #将s结点链接到rear结点后面
        self.rear=s
```





3.2.4 队列的链式存储结构及其基本运算算法实现

(3) 出队pop()



```
def pop(self):
```

```
    assert not self.empty()
```

```
    if self.front == self.rear:
```

```
        e = self.front.data
```

```
        self.front = self.rear = None
```

```
    else:
```

```
        e = self.front.data
```

```
        self.front = self.front.next
```

```
    return e
```

#出队操作

#检测空链队

#原链队只有一个结点

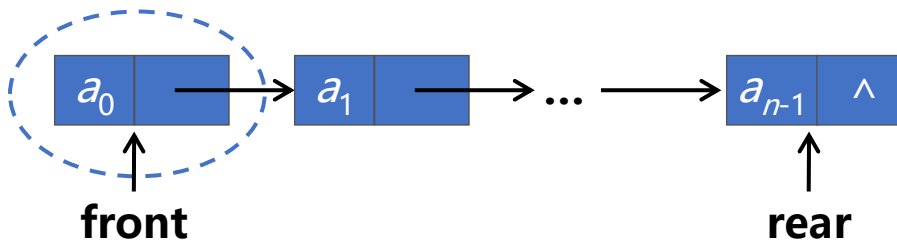
#取首结点值

#置为空队

#原链队有多个结点

#取首结点值

#front指向下一个结点

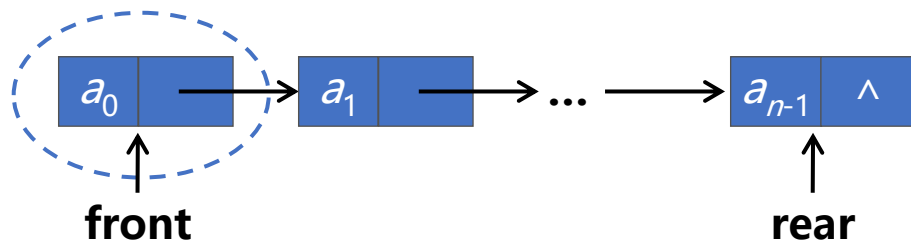




3.2.4 队列的链式存储结构及其基本运算算法实现

(4) 取队头元素gethead()

```
def gethead(self):           #取队头元素
    assert not self.empty()  #检测空链队
    e=self.front.data        #取首结点值
    return e
```





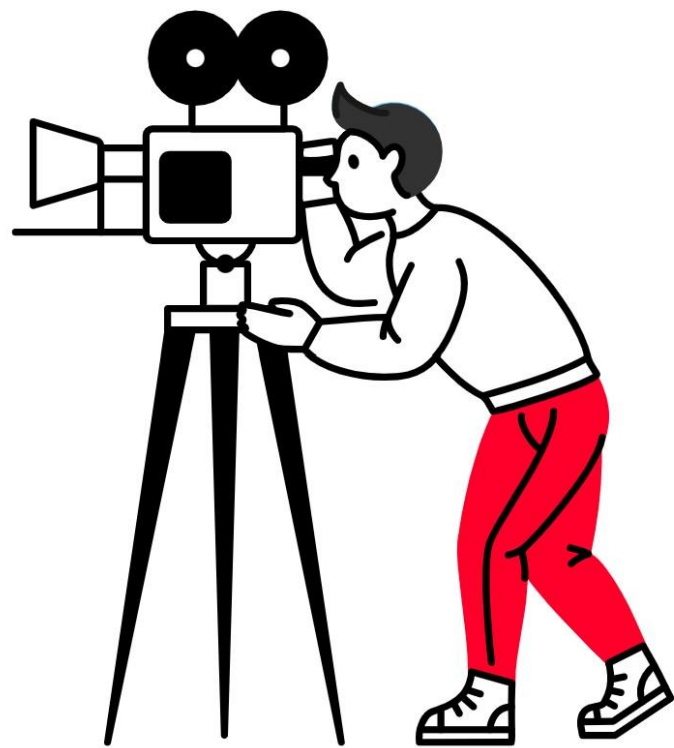
3.2.5 链队的应用算法设计示例

【例3.13】 采用链队求解第2章例2.16的约瑟夫问题。



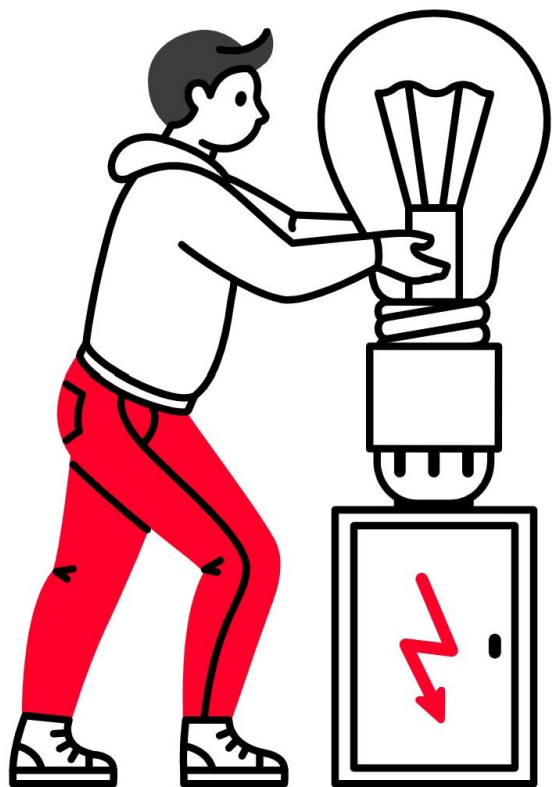
【例2.16】 编写一个程序求解约瑟夫 (Joseph) 问题。有 n 个小孩围成一圈，给他们从1开始依次编号，从编号为1的小孩开始报数，数到第 m 个小孩出列，然后从出列的下一个小孩重新开始报数，数到第 m 个小孩又出列，...，如此反复直到所有的小孩全部出列为止，求整个出列序列。

如当 $n=6$ ， $m=5$ 时的出列序列是5，4，6，2，3，1。





3.2.5 链队的应用算法设计示例



解：先定义一个链队qu:

- 对于 (n, m) 约瑟夫问题，依次将 $1 \sim n$ 进队。
- **循环n次出列n个小孩**：依次出队 $m-1$ 次，将所有出队的元素立即进队（将他们从队头出队后插入到队尾），再出队第 m 个元素并且输出（出列第 m 个小孩）。



3.2.5 链队的应用算法设计示例



```
from LinkQueue import LinkQueue
def Jsequence(n,m):           #求约瑟夫序列
    qu=LinkQueue()           #定义一个链队
    for i in range(1,n+1):    #进队编号为1到n的n个小孩
        qu.push(i)
    for i in range(1,n+1):    #共出列n个小孩
        j=1
        while j<=m-1:        #出队m-1个小孩，并将他们进队到队尾
            qu.push(qu.pop())
            j+=1
        x=qu.pop()            #出队第m个小孩
        print(x,end=' ')
    print()
```



3.2.5 链队的应用算法设计示例

#主程序

print()

print(" 测试1: n=6,m=3")

print(" 出列顺序:",end=' ')

Jsequence(6,3)

print(" 测试2: n=8,m=4")

print(" 出列顺序:",end=' ')

Jsequence(8,4)

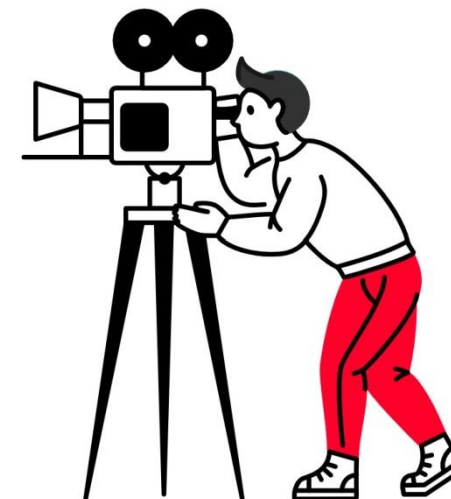


```
管理员: C:\windows\system32\cmd.exe
D:\Python\ch3\示例>python exam3-13.py
测试1: n=6,m=3
出列顺序: 3 6 4 2 5 1
测试2: n=8,m=4
出列顺序: 4 8 5 2 1 3 7 6
D:\Python\ch3\示例>
```



3.2.6 Python中的双端队列deque

- 双端队列是在队列基础上扩展而来的，其示意图如下图所示。
- 双端队列与队列一样，元素的逻辑关系也是线性关系，但队列只能在一端进队，另外一端出队，而双端队列可以在两端进行进队和出队操作，具有队列和栈的特性，因此使用更加灵活。





3.2.6 Python中的双端队列deque



- **输入受限的双端队列**



- **输出受限的双端队列**





3.2.6 Python中的双端队列deque

Python提供了一个集合模块collections，里面封装了多个集合类，其中包括deque即双端队列（double-ended queue）。

1. 创建双端队列

```
qu=deque()           #创建一个空的双端队列qu
qu=deque(maxlen=N)    #创建一个固定长度为N的双端队列qu
qu=deque(L)           #创建的双端队列qu中包含列表L中的元素
```





3.2.6 Python中的双端队列deque

2. 双端队列的方法

- `deque.clear()`: 清除双端队列中的所有元素。
- `deque.append(x)`: 在双端队列的右端添加元素`x`。时间复杂度为 $O(1)$ 。
- `deque.appendleft(x)`: 在双端队列的左端添加元素`x`。时间复杂度为 $O(1)$ 。
- `deque.pop()`: 在双端队列的右端出队一个元素。时间复杂度为 $O(1)$ 。
- `deque.popleft()`: 在双端队列的左端出队一个元素。时间复杂度为 $O(1)$ 。
- `deque.remove(x)`: 在双端队列中删除首个和`x`匹配的元素（从左端开始匹配的），如果没有找到抛出异常。时间复杂度为 $O(n)$ 。
- `deque.count(x)`: 计算双端队列中元素为`x`的个数。时间复杂度为 $O(n)$ 。
- `deque.extend(L)`: 在双端队列的右端添加列表`L`的元素。例如，`qu`为空，`L=[1, 2, 3]`，执行后`qu`从左向右为`[1, 2, 3]`。
- `deque.extendleft(L)`: 在双端队列的左端添加列表`L`的元素。例如，`qu`为空，`L=[1, 2, 3]`，执行后`qu`从左向右为`[3, 2, 1]`。
- `deque.reverse()`: 把双端队列里的所有元素的逆置。
- `deque.rotate(n)`: 双端队列的移位操作，如果`n`是正数，则队列所有元素向右移动`n`个位置，如果是负数，则队列所有元素向左移动`n`个位置。

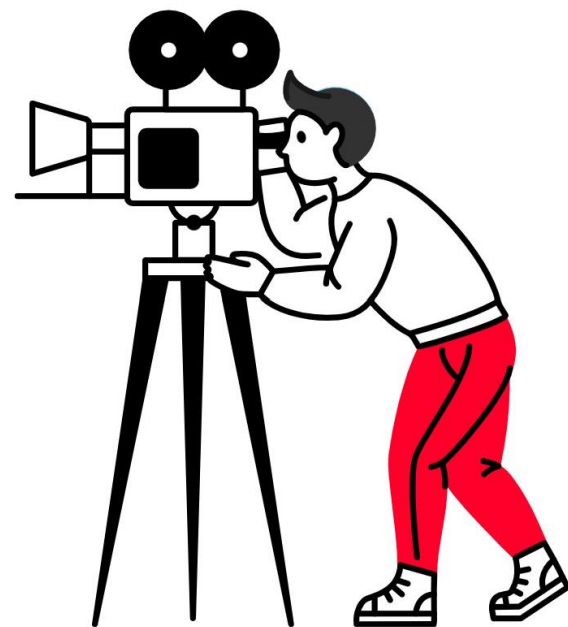


3.2.6 Python中的双端队列deque

3.用双端队列实现栈

- 以左端作为栈底（左端保持不动），右端作为栈顶（右端动态变化，`st[-1]`为栈顶元素），栈操作在右端进行，则用`append()`作为进栈方法，`pop()`作为出栈方法。
- 以右端作为栈底（右端保持不动），左端作为栈顶（左端动态变化，`st[0]`为栈顶元素），栈操作在左端进行，则用`appendleft()`作为进栈方法，`popleft()`作为出栈方法。

```
from collections import deque    #引用deque
st=deque()
st.append(1)
st.append(2)
st.append(3)
while len(st)>0:
    print(st.pop(),end=' ')    #输出: 3 2 1
print()
```





3.2.6 Python中的双端队列deque

4.用双端队列实现普通队列

- 以左端作为队头（出队端，），右端作为队尾（进队端），则用popleft()作为出队方法，append()作为进队方法。在队列非空时qu[0]为队头元素，qu[-1]为队尾元素。
- 以右端作为队头（出队端），左端作为队尾（进队端），则用pop()作为出队方法，appendleft()作为进队方法。在队列非空时qu[-1]为队头元素，qu[0]为队尾元素。

```
from collections import deque
```

```
qu=deque()
```

```
qu.append(1)
```

```
qu.append(2)
```

```
qu.append(3)
```

```
while len(qu)>0:
```

```
    print(qu.popleft(),end=' ')
```

```
print()
```

#输出: 1 2 3

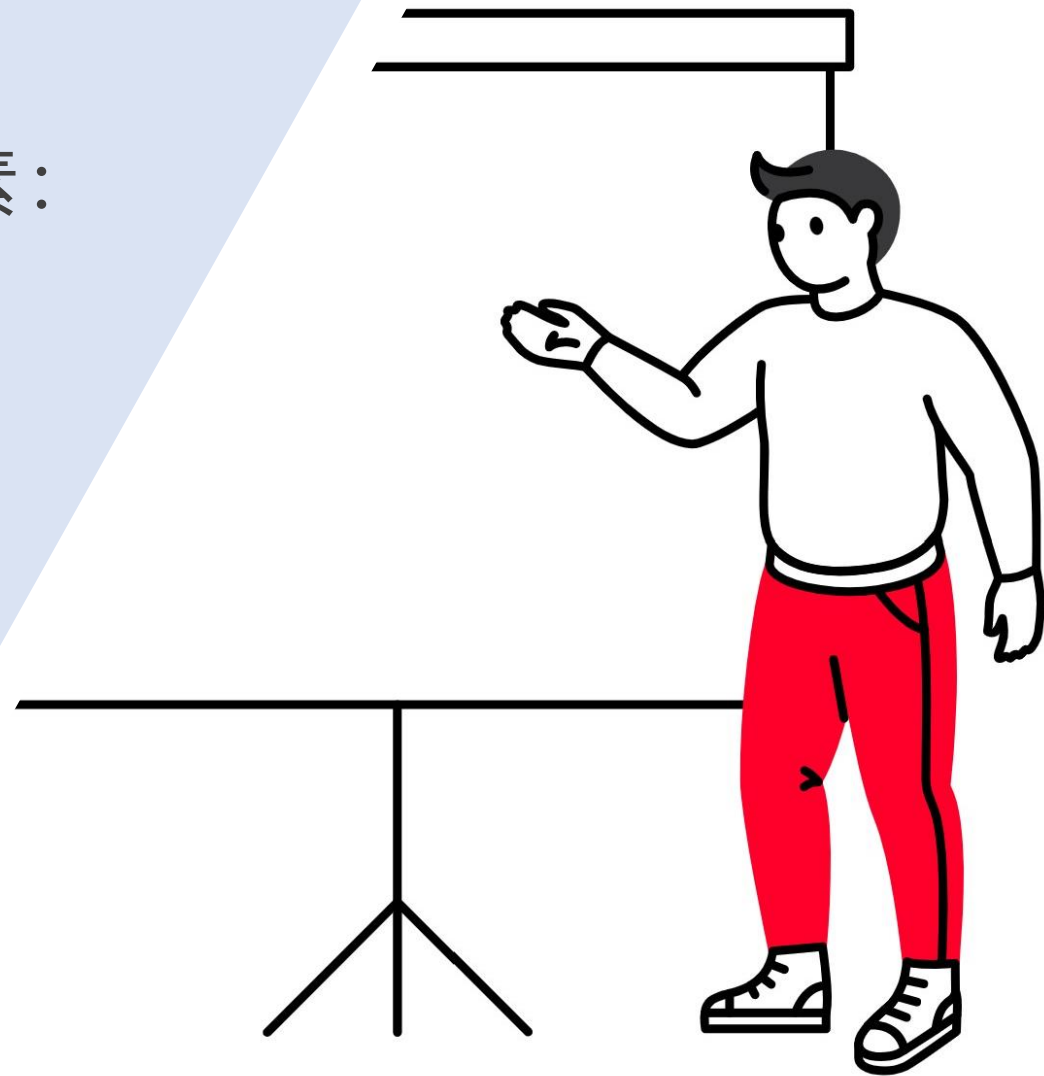




3.2.7 队列的综合应用

求解问题中需要临时保存一些数据元素：

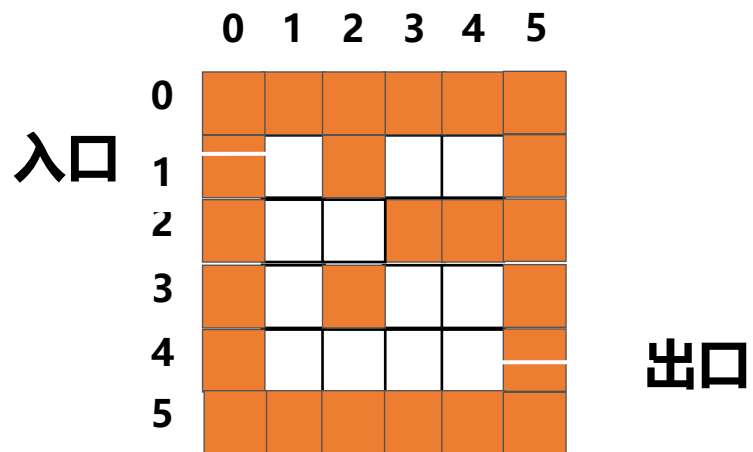
- 先保存的后处理：栈
- 先保存的先处理：队列





3.2.7 队列的综合应用

求迷宫问题

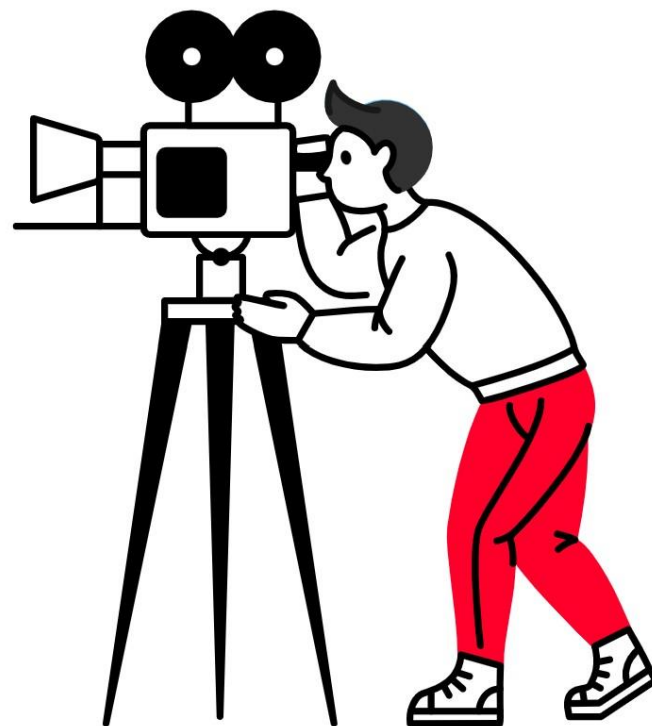


一个迷宫图



求从入口到出口的一条简单路径

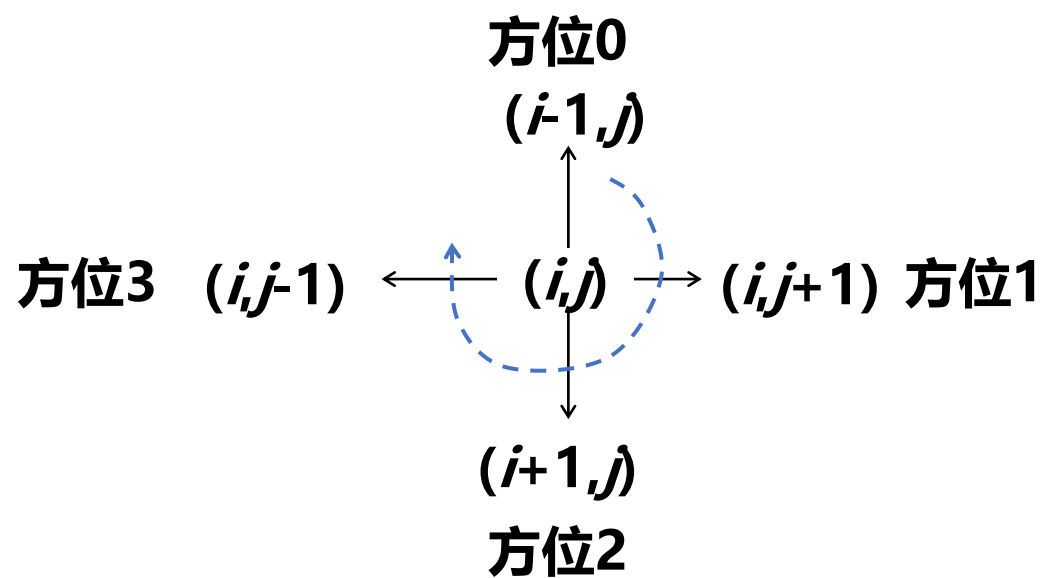
```
int [][] mg=  
{ {1,1,1,1,1,1},  
  {1,0,1,0,0,1},  
  {1,0,0,1,1,1},  
  {1,0,1,0,0,1},  
  {1,0,0,0,0,1},  
  {1,1,1,1,1,1}  
};
```





3.2.7 队列的综合应用

试探顺序



$dx = [-1, 0, 1, 0]$

$dy = [0, 1, 0, -1]$

#x方向的偏移量

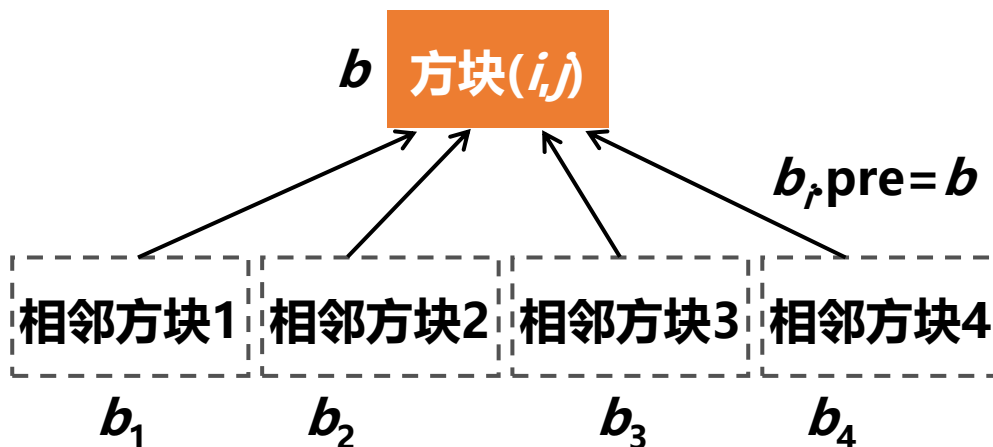
#y方向的偏移量





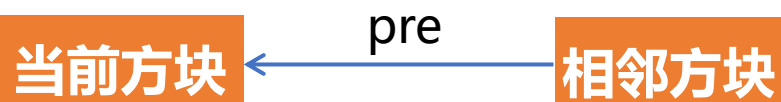
迷宫问题的搜索过程

- 每次走到一个方块 (i,j) ，一次性试探所有相邻方块，将所有相邻可走方块进队。
- 一个方块在队列中的元素为 b ，并保存其前驱方块（pre标识）。
- 从入口开始，找到出口后由pre推导出迷宫路径。





3.2.7 队列的综合应用



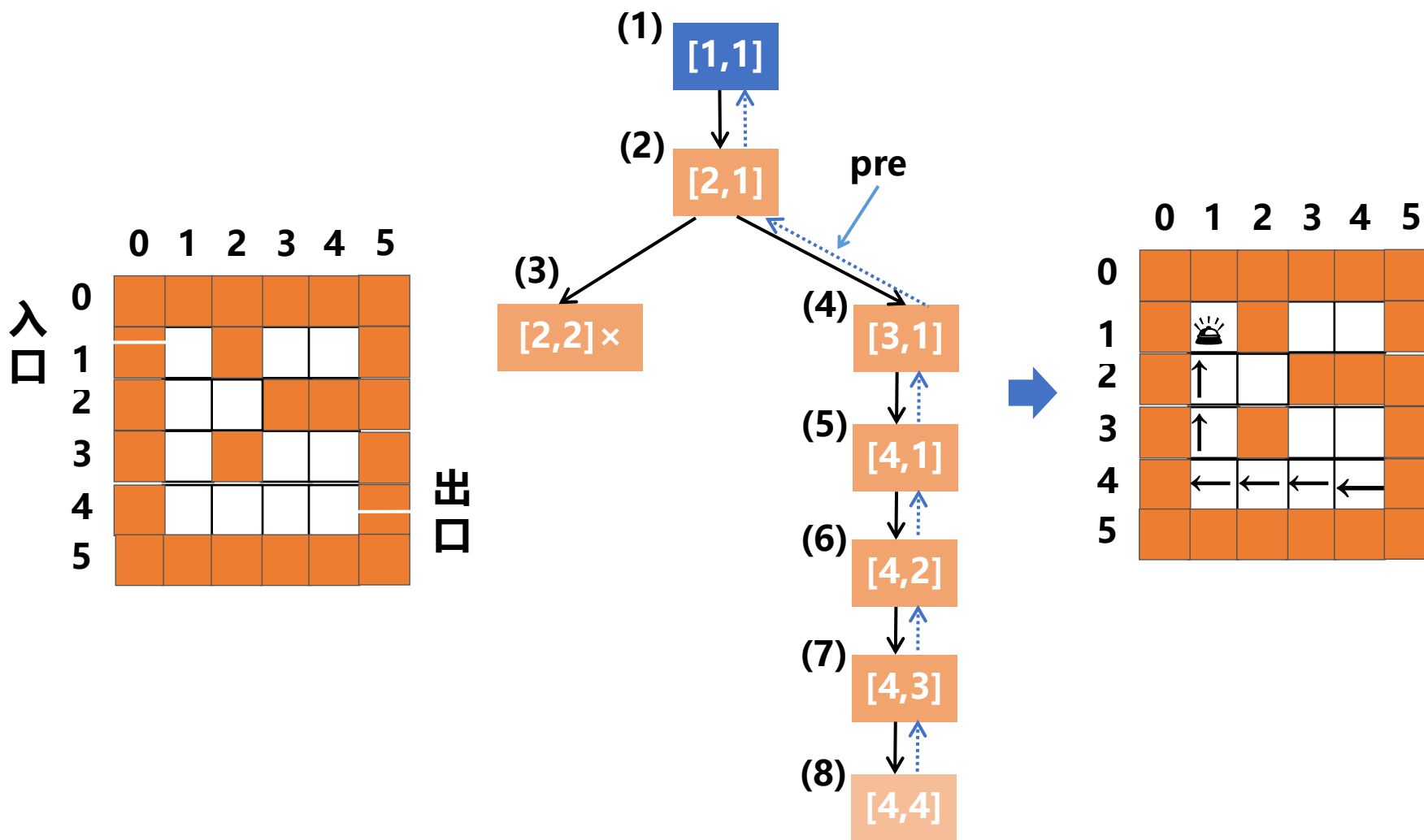
```
class Box:
    def __init__(self,i1,j1):
        self.i=i1
        self.j=j1
        self.pre=None
```

#方块类
#构造方法
#方块的行号
#方块的列号
#前驱方块





3.2.7 队列的综合应用

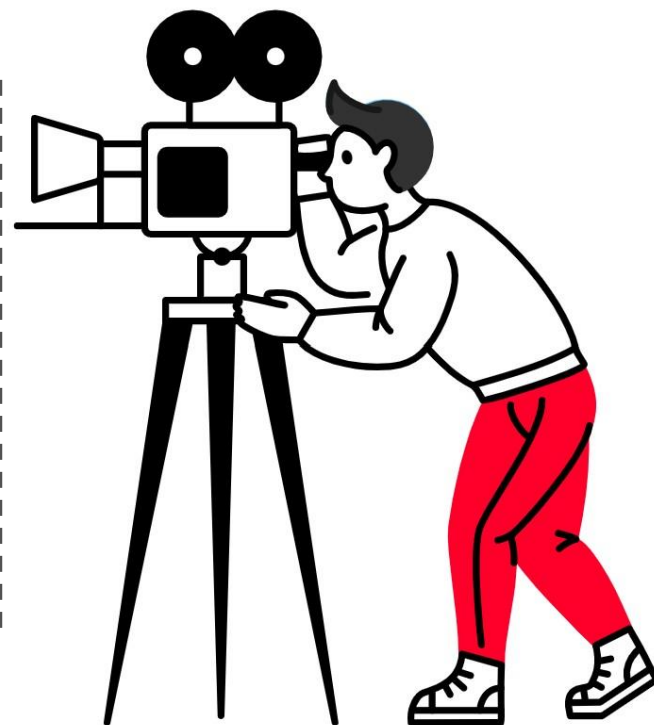




3.2.7 队列的综合应用

```
def mgpath(xi,yi,xo,ye):  
    global mg  
    dx=[-1,0,1,0]  
    dy=[0,1,0,-1]  
    qu=deque()  
    b=Box(xi,yi)  
    qu.appendleft(b)  
    mg[xi][yi]=-1
```

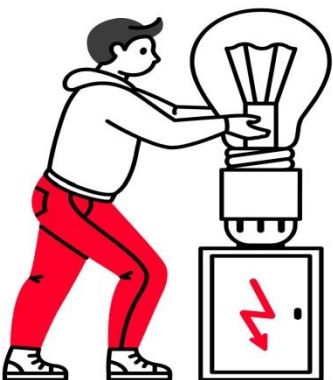
#求(xi,yi)到(xo,ye)的一条迷宫路径
#迷宫数组为全局变量
#x方向的偏移量
#y方向的偏移量
#定义一个队列
#建立入口结点b
#结点b进队
#进队方块mg值置为-1





3.2.7 队列的综合应用

```
while len(qu)!=0:                                #队不空时循环
    b=qu.pop()                                    #出队一个方块b
    if b.i==xe and b.j==ye:                        #找到了出口,输出路径
        p=b                                       #从b出发回推导出迷宫路径并输出
        path=[]                                  #path存放逆路径
        while p!=None:                            #找到入口为止
            path.append "[" + str(p.i) + "," + str(p.j) + "]"
            p=p.pre
        for i in range(len(path)-1,-1,-1):        #反向输出path得到正向路径
            print(path[i],end=' ')
    return True                                  #找到一条路径时返回True
```





3.2.7 队列的综合应用

```
for di in range(4): #循环扫描每个相邻方位的方块
    i,j=b.i+dx[di],b.j+dy[di] #找b的di方位的相邻方块(i,j)
    if mg[i][j]==0: #找相邻可走方块
        b1=Box(i,j) #建立后继方块结点b1
        b1.pre=b #设置其前驱方块为b
        qu.appendleft(b1) #b1进队
        mg[i][j]=-1 #进队的方块置为-1
return False #未找到任何路径时返回False
```

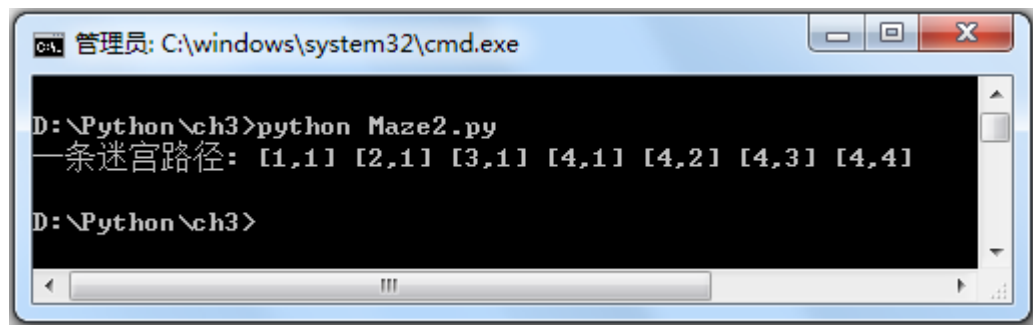




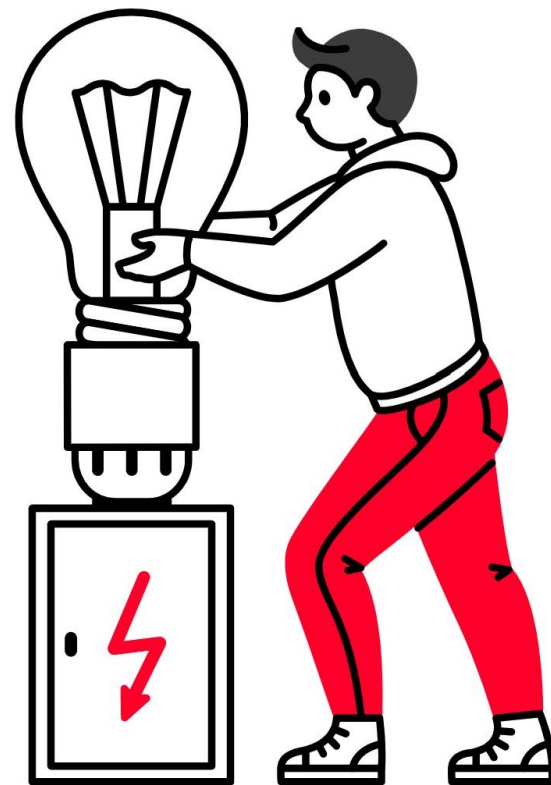
3.2.7 队列的综合应用

设计主程序

```
xi,yi=1,1  
xe,ye=4,4  
print("一条迷宫路径:",end=' ')  
if not mgpath(xi,yi,xe,ye):    #(1,1)->(4,4)  
    print("不存在迷宫路径")  
print()
```



	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						





3.2.7 队列的综合应用



”

为什么用队列找到的路径一定是最短路径？

3.2.8 优先队列

01

优先队列就是指定队列中元素的优先级，按优先级越大越优先出队，而普通队列中按进队的先后顺序出队，可以看成**进队越早越优先**。

02

优先队列按照根的大小分为**大根堆**和**小根堆**，大根堆的元素**越大越优先出队**（即元素越大优先级也越大），小根堆的元素**越小越优先出队**（即元素越小优先级也越大）。





3.2.8 优先队列

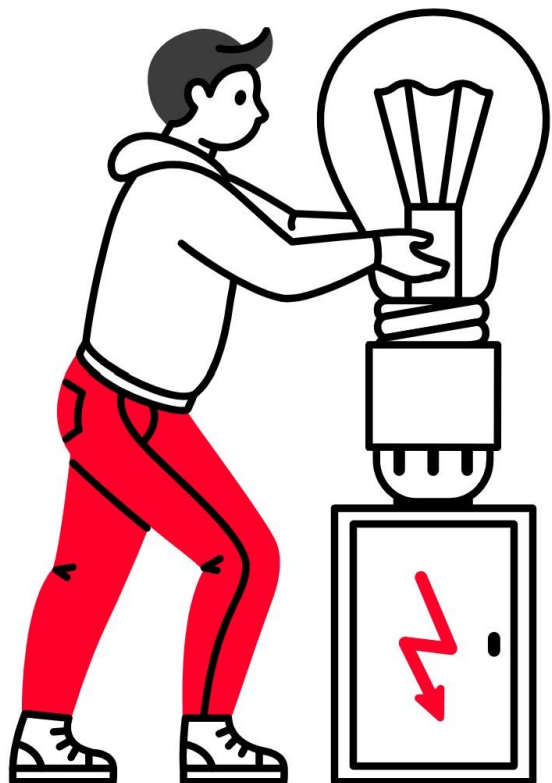
Python中提供了**heapq**模块，其中包含堆的基本操作方法用于创建堆，但只能创建小根堆。其主要方法如下：

- `heapq.heapify(list)`：把列表list调整为堆。
- `heapq.heappush(heap, item)`：向堆heap中插入元素item（进队item元素），该方法会维护堆的性质。
- `heapq.heappop(heap)`：从堆heap中删除最小元素并且返回该元素值。
- `heapq.heapreplace(heap, item)`：从堆heap中删除最小元素并且返回该元素值，同时将item插入并且维护堆的性质。它优于调用函数`heappop(heap)`和`heappush(heap, item)`。
- `heapq.heappushpop(heap, item)`：把元素item插入到堆heap中，然后从heap中删除最小元素并且返回该元素值。它优于调用函数`heappush(heap, item)`和`heappop(heap)`。
- `heapq.nlargest(n, iterable[, key])`：返回迭代数据集合iterable中第n大的元素，可以指定比较的key。它比通常计算多个list第n大的元素方法更方便快捷。
- `heapq.nsmallest(n, iterable[, key])`：返回迭代数据集合iterable中第n小的元素，可以指定比较的key。它比通常计算多个list第n小的元素方法更方便快捷。
- `heapq.merge(*iterables)`：把多个堆合并，并返回一个迭代器。



3.2.8 优先队列

例如，定义一个heapq列表，将其调整为小根堆，调用一系列heapq方法及其输出结果如下：



```
import heapq
heap=[6,5,4,1,8]
heapq.heapify(heap)
print(heap)
heapq.heappush(heap,3)
print(heap)
print(heapq.heappop(heap))
print(heap)
print(heapq.heapreplace(heap,2))
print(heap)
print(heapq.heappushpop(heap,1))
print(heap)
```

#定义一个列表heap
#将heap列表调整为堆
#输出:[1,5,4,6,8]
#进队3
#输出:[1,5,3,6,8,4]
#输出:1
#输出:[3,5,4,6,8]
#输出:3(出队最小元素, 再插入2)
#输出:[2,5,4,6,8]
#输出:1(插入1, 再出队最小元素)
#输出:[2,5,4,6,8]



3.2.8 优先队列

- 由于heapq不支持大根堆，那么如何创建大根堆呢？
- 对于数值类型，一个最大数的相反数就是最小数，可以通过对数值取反、仍然创建小根堆的方式来获取最大数。





3.2.8 优先队列

凌晨4点半的哈佛大学图书馆实况

